CoopRT: Accelerating BVH Traversal for Ray Tracing via Cooperative Threads

Yavuz Selim Tozlu
ystozlu@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Huiyang Zhou
hzhou@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

Ray Tracing is a rendering technique to simulate the way light interacts with objects to create realistic images. It has become prominent thanks to the latest hardware support on Graphics Processing Units (GPUs), i.e., the Ray-Tracing (RT) unit, specially designed to accelerate ray tracing operations. Despite such hardware advances, ray tracing remains a performance bottleneck for high-performance graphics workloads, such as real-time path tracing (PT), which is an application of ray tracing where multiple bouncing rays are traced per pixel. The key reasons are (a) the costly Bounding Volume Hierarchy (BVH) traversal operation, and (b) low Single-Instruction-Multiple-Thread (SIMT) efficiency as the rays in the same warp deviate inevitably in their traversal paths.

In this work, we propose a novel architecture design for cooperative BVH traversal that exploits the parallelism present in the BVH traversal process. The key idea of our CoopRT scheme is to make use of the idle threads, either completely inactive when the ray tracing instruction is executed or partially idle due to early completion, to help the long running threads in the same warp. Specifically, we enable idle threads in a GPU warp to utilize their readily available traversal hardware to help traverse the BVH tree for the busy threads, therefore helping them finish their traversal much faster. This approach is implemented purely in hardware, requiring no changes to the programming model. We present our architecture design and show that it only involves small changes to the existing RT unit.

We evaluated CoopRT in Vulkan-sim, a cycle-level simulator, and observed up to 5.11x speedup over the baseline, with a geometric mean of 2.15x speedup at the cost of a moderate area overhead of 3.0% of the warp buffer in the RT unit. Using the energy-delay product, our CoopRT achieves an average of 2.29x improvement over the baseline.

CCS Concepts

• **Computing methodologies** → *Graphics systems and interfaces*; • **Computer systems organization** → **Multicore architectures**; **Single instruction, multiple data**.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731118

Keywords

Ray tracing, GPU, 3D graphics

ACM Reference Format:
Yavuz Selim Tozlu and Huiyang Zhou. 2025. CoopRT: Accelerating BVH Traversal for Ray Tracing via Cooperative Threads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731118

1 Introduction

Ray tracing (RT) is a 3D rendering technique that can achieve life-like graphics by simulating how light rays travel through 3D scenes. It has been used in offline rendering for movies and animations [16]. With the special hardware support available in GPUs now, ray tracing is also being used in real-time applications such as video games [1][17]. Modern game engines adopt various ray tracing algorithms, such as ray-traced shadows and reflections [6][7], which can be used in conjunction with traditional rasterization techniques to enhance visual realism. Beyond computer graphics, ray tracing is also used in various fields such as wireless communication for channel estimation, propagation modeling, and 3D imaging [20][27][28][39][43].

In practice, ray tracing involves traversing a tree-like structure called the Bounding Volume Hierarchy (BVH) tree [25]. A 3D scene is encoded as a hierarchical arrangement of Axis-Aligned Bounding Boxes (AABBs), where objects are grouped based on their proximity and size. These groups are enclosed in AABBs, and the collection of AABBs forms the BVH tree. Rays are modeled as 3D vectors, which traverse the BVH tree and test intersection against the AABBs to find the closest-hit primitive, a basic geometric shape such as a triangle, quad, or sphere.

The BVH tree can become large depending on the scene's complexity and often exceeds the capacity of on-chip caches. A traversal involves reading BVH nodes, performing intersections, and processing the subsequent nodes based on the intersection results. In large scenes, these memory accesses often miss in the caches and require expensive DRAM accesses, during which the threads stall and wait for the data to return. In addition, each thread traces a ray with different directions, causing the rays to diverge and traverse different portions of the BVH tree. As a result, the traversal process can be slow and memory-bound, making it a primary bottleneck in ray tracing performance.

Latest consumer-grade GPUs incorporate a specialized hardware unit called the RT unit to accelerate ray tracing and enable real-time performance [3][4][5]. The RT unit is triggered when a warp encounters the specialized *trace_ray* instruction. The semantics of this instruction is that each thread in the warp traces a ray by

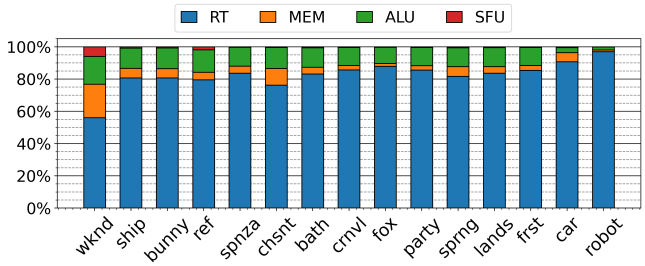


Figure 1: Pipeline stalls from different instructions, RT: *trace_ray* instructions, MEM: load/store instructions from CUDA cores, ALU: compute instructions from CUDA cores, SFU: special function instructions from CUDA cores. Configuration: 256x256 resolution, path tracing, 1 sample-per-pixel.

traversing the BVH tree and performing intersection tests to find the closest-hit primitive, if one exists. A warp of 32 threads traces up to 32 rays if all the threads are active.

Despite the hardware support for ray tracing, performing real-time ray tracing, especially path tracing, is still very challenging. The key reasons are (1) for a high-resolution image, the number of rays to be traced is very high; (2) BVH traversals are memory-bound, dominated by reading tree nodes, with little computation needed mainly for intersection tests and coordinate transformations; and (3) there are high degrees of divergence as different rays follow different traversal paths and bounce in different directions.

In Fig. 1, we present the contribution of pipeline stalls from different types of instructions across various scenes in a ray tracing application. It can be seen that most of the stalls are due to the *trace_ray* instructions.

To reveal the performance issues with the *trace_ray* instructions, we look into path tracing (PT), which is a specific application of ray tracing to render a 2D image of a 3D scene [13][14]. With PT, a primary ray is cast for each pixel, bouncing through the scene for a set number of times or until it hits a light source or escapes the scene. Each bounce contributes to the final color of the pixel and involves a *trace_ray* instruction (more details in Section 3). As the rays bounce through the scene with different paths, their traversing behaviors diverge. Given the SIMD nature of a warp, such divergence results in two levels of resource under-utilization. Firstly, as a ray misses/escapes the scene, the corresponding thread becomes inactive for subsequent bounces or subsequent *trace_ray* instructions. As a result, when a *trace_ray* instruction is dispatched to the RT unit, some of the threads may be completely inactive or idle. As the active threads continue tracing the bouncing rays through the scene, the number of idle threads in a warp typically increases as the number of bounces increases. Fig. 2 shows how the ratio of active threads in a warp on average changes over time for various scenes on the baseline RT unit. We can see that all the warps have 100% SIMT efficiency (i.e., 100% active threads) when tracing the primary rays at the beginning. After only a few bounces, the efficiency starts to drop significantly. Secondly, among active threads, some may finish their traversal earlier than others, thereby being partially idle during the *trace_ray* instruction execution.

With the observation that there exists significant divergence and the resulting resource under-utilization during ray tracing, we

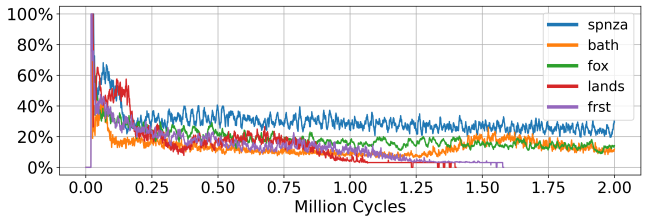


Figure 2: Percent of threads that are busy in RT unit.

propose a novel way to overcome this fundamental performance challenge. Our idea is to parallelize BVH traversal by employing the abundant idle threads. The key insight is for a single ray, in order to determine its closest-hit in the BVH tree, the traversal can be effectively parallelized without error. In other words, the workload of one active thread in a warp in the RT unit performing the BVH tree traversal can be parallelized and distributed among multiple threads. Considering that each thread already has dedicated traversal hardware in the RT unit, if we enable the idle threads to access the active threads' traversal stacks, the idle threads can traverse the BVH as usual, i.e., processing the nodes, and pushing new nodes into their own stacks based on the intersection test results. In our proposed scheme, the cooperation of threads is done completely in the RT unit hardware and is transparent to the software. Therefore, the cooperative execution only changes how a *trace_ray* instruction is executed in the RT unit and does not affect rest of the GPU hardware or software.

In summary, this paper makes the following contributions,

- We propose a cooperative BVH traversal algorithm that utilizes existing GPU hardware to parallelize BVH traversal and accelerate ray tracing with no changes to the programming model.
- We propose the architecture design for cooperative BVH traversal.
- We model the cooperative BVH traversal in Vulkan-sim, a cycle-level simulator, to evaluate its performance and show that our CoopRT scheme achieves up to 5.11x with a geometric mean of 2.15x speedup.

2 Background

2.1 Ray Tracing and BVH Traversal

Ray tracing is a 3D computer graphics method that can produce highly realistic visuals. It simulates light rays through a 3D scene and calculates how much illumination the objects would receive. Rays are modeled as 3D vectors with an origin, usually the camera, and a direction through the pixels on the 2D image plane. These rays are tested against the primitives in the scene to find the *closest-hit* or detect if there is *any-hit* at all. This can be used to accurately compute global illumination, shadows, reflections, or any lighting effect in a scene. Therefore, ray tracing is commonly used together with rasterization to augment the lighting effects, and still maintain real-time performance.

Ray tracing can also be used exclusively to render 2D images of 3D scenes with a technique known as path tracing (PT) [30]. In PT, primary rays originating from the camera are traced until they either miss the scene or hit an object. Upon hitting an object, the

direction of reflection is calculated, and subsequent bouncing rays are traced. This bouncing process continues until a preset number of times determined by the programmer, 16 in this study, or until the rays hit a light source or miss the scene. Each bounce contributes to the pixels' final color.

Objects in a 3D scene are usually modeled as a collection of triangles. While it is possible to test rays against every triangle in the scene to find the closest-hit object, this quickly becomes impractical due to the large number of triangles in a scene. To accelerate ray tracing, BVH is utilized to encode the 3D scene in a tree-like structure of AABBs, where the root node is an AABB that bounds the whole scene [23][25][29][36]. When traversing a BVH tree, if a ray does not intersect an AABB, then it is guaranteed that the ray will not intersect any primitives inside the AABB, therefore they can be skipped.

BVH can be built on the CPU or the GPU using various methods such as the surface area heuristic [32]. The traversal starts from the root node and typically follows a depth-first search (DFS). In DFS, a traversal *stack* is used to track nodes. The traversal stack stores the addresses of the nodes that will be accessed afterwards. A BVH tree node can be either an internal node or a leaf node. Internal nodes have child nodes, and they contain the coordinates and addresses of their children. Leaf nodes are primitives such as triangles or quads, and they contain the vertex coordinates of the primitive. Traversal involves popping nodes from the stack and fetching them from memory. For internal nodes, their children are tested for intersections and pushed onto the stack if a hit is detected. For leaf nodes, the primitives are directly tested for intersections. Traversal continues until the stack is empty, or any-hit is found, based on the criteria defined by the programmer according to the application's needs.

High-performance ray tracing applications are typically run on GPUs. Modern graphics application programming interfaces (API), such as Vulkan [41] or DirectX, feature sophisticated ray tracing pipelines that programmers can use to develop ray tracing applications. The ray tracing pipeline introduces programmable shader stages such as *raygen*, *closest-hit*, *miss*, and optionally *intersection* and *any-hit*. Rays are generated in the *raygen* shader using the *trace_ray* instruction. If a hit primitive is found, *closest-hit* shader is invoked to carry out reflection/illumination calculations, otherwise; the *miss* shader is invoked. Custom geometry, such as spheres, can be defined using the *intersection* shader. The *any-hit* shader is executed for each potential intersection, allowing for effects like transparency by conditionally accepting or discarding hits along the ray's path.

In GPU ray tracing applications, the BVH is built by the GPU driver. In this work, the open-source ray tracing library Embree 3.14 [2] is used to build BVH trees for Vulkan-sim.

2.2 GPU Architecture

GPUs are throughput oriented massively parallel computers. They consist of thousands of hardware threads that can run in parallel [31]. Fig. 3 shows a generic GPU architecture which we consider as the baseline. A GPU contains an array of Streaming Multiprocessors (SM), as in Nvidia terminology. Each SM has a collection of execution lanes, register files, warp schedulers, and dedicated L1 caches. Shader programs are executed with a thread hierarchy of thread blocks (TBs), each composed of warps, with a warp of 32 threads that execute in a lock-step manner. TBs are assigned to SMs by the Gigathread Engine [40]. GPUs hide memory latencies by employing fine-grained multithreading. When a warp stalls, the warp scheduler issues instructions from another non-stalling warp instead. As long as there are enough warps, long instruction execution latencies, such as memory latency, can be hidden by continuously scheduling non-stalling warps. Modern GPUs can have anywhere from fewer than 10 (Mobile) to over 100 (Desktop) SMs, depending on the use case. Typically, there is an L2 cache shared among all SMs, connected via a crossbar. For off-chip memory, high-bandwidth DRAM modules are used with on-chip memory controllers.

2.3 Hardware Support for Ray Tracing

Latest GPUs feature specialized hardware called the RT Unit/Core to efficiently perform ray tracing [4][5]. Vulkan-sim is a GPU architectural simulator that models an RT unit, which we use to model our proposed CoopRT in this study. The RT unit can be viewed as a specialized execution lane operating at warp granularity. When a warp encounters the *trace_ray* instruction, the instruction is steered to the RT unit, where the hardware performs BVH traversals. Specifically, each thread of the warp is assigned a ray and traverses the BVH tree to find either the closest-hit or any-hit primitive. As shown in Fig. 3, an RT unit has a warp buffer which keeps the ray data and traversal stack for each thread. At each cycle, a warp from the warp buffer is selected, and a memory access request from that warp is served. When the data returns, an intersection test is performed by the operation units. Based on the result, either the node's children are pushed to the traversal stack (upon a hit), or if it is a primitive, the closest-hit value is updated if needed. The per-thread traversal stack stores the addresses of the nodes instead of the node data itself. While the RT unit does not reduce the latency of memory requests, it accelerates the traversal by streamlining the address calculations and intersection tests.

3 Thread Activity In Ray Tracing

The latency of ray tracing applications is usually dominated by the CISC-like *trace_ray* instruction, which traverses the BVH tree, performs intersection tests and writes the results back to memory. Listing 1 shows the anatomy of a *raygen* shader. When compiled, the *traceRay()* function expands to a block of code that includes the performance-dominating *trace_ray* instruction, followed by control flow instructions that branch off to the *closest-hit*, *any-hit*, *intersection*, or *miss* shaders. As the listing shows, each iteration of the loop processes one bounce and invokes a trace_ray instruction. Like any GPU shaders, the *raygen* shader is executed in the SIMT manner. Due to such SIMT execution, there are two sources of hardware resource under-utilization: **inactive** threads and **early finishing** threads. With rays bouncing further into the scene every iteration, more threads become inactive as they miss the scene or hit a light source and exit the loop. However, although some threads become inactive, as long as there is at least one active thread in the warp, the loop keeps repeating. Inactive threads are masked off in hardware and they do not perform any traversal, leading to unused hardware.

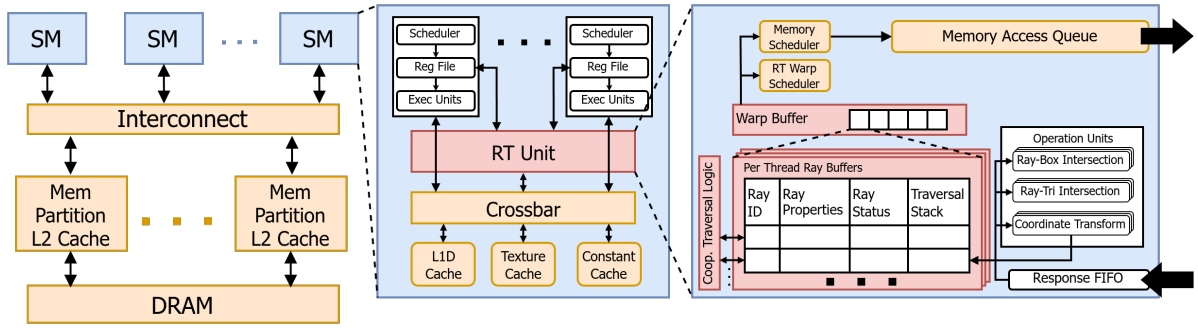


Figure 3: Diagram of the GPU model used in this study. Red blocks indicate the modified components. Redrawn from [37].

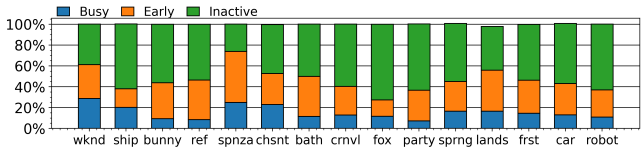


Figure 4: Thread status distribution.

In addition, the execution latency of the *trace_ray* instruction itself is variable among threads, similar to load instructions where some threads hit in the cache, while others miss. In the case of *trace_ray* instruction, this latency variation is much more severe, as some threads might quickly find the closest-hit primitive or just miss the scene, whereas other threads might spend a long time traversing the BVH to find their closest-hit primitive. We refer to the threads in a warp that finish earlier than the others as early finishing threads.

To quantify the importance of inactive and early finishing threads, we simulate the baseline GPU in Vulkan-sim and gather thread status data at fixed intervals and average them. Fig. 4 shows the distribution of thread status across different 3D scenes. From the figure, we can see a high number of threads spend most of their time being inactive, or finishing early and waiting.

Note that the divergence behavior in ray tracing (i.e., the *raygen* shader and the *trace_ray* instruction) differs from typical GPGPU applications, where the programs have relatively more balanced if-then-else paths. In comparison, as shown in Listing 1, the majority of the execution time is spent inside the loop, leaving little to no work for threads that exit the loop. Fig. 5 shows a simplified Control Flow Graph for the program in Listing 1. Existing SIMT control flow handling techniques could be beneficial for this code to some extent [18][19][21][22][42]. Dynamic Warp Formation[22] and Thread Block Compaction[21] could be used to create new warps when threads diverge to blocks $C0$, $C1$ and M. Similarly, multi-path execution[19] could execute these blocks in parallel. However, when threads diverge at block L to T and S, these techniques become insufficient because the latency of S is negligible compared to block T, which contains the *trace_ray* instruction. Therefore, the existing techniques can mitigate some of the divergence in ray tracing, but none of them address the main bottleneck, which is the BVH traversal process, i.e., block T.

Different from the existing schemes, CoopRT offers a novel way to exploit the inactive or early finishing threads to parallelize and

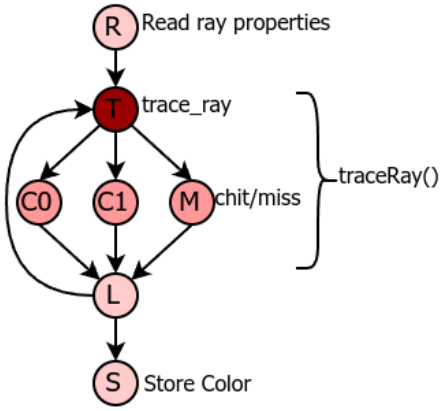


Figure 5: A Simplified Control Flow Diagram of Listing 1. $C0$, $C1$ are closest-hit shaders, and M is a miss shader. L is the loop iteration block.

accelerate the BVH traversal process, therefore improving the resource utilization and significantly reducing the latency of the *trace_ray* instruction and the *raygen* shader. More generally, as each *trace_ray* instruction essentially performs 32 DFS operations and the *raygen* shader can be viewed as a sequence of dependent DFS operations, CoopRT provides a novel way to accelerate such DFS operations, which has more profound impacts when the RT unit is repurposed for accelerating graph algorithms [11][26][44].

Listing 1: Simplified raygen shader for path tracing.

```
//Calculate ray origin and direction
//using pixel coordinates
for(int i=0;i<NUM_BOUNCES;i++){
  traceRay(ray.orig, ray.dir)
  if missed || !scattered
    break;
  //Calculate new origin and direction
  //using the hit data
}
```

```
//Calculate and store pixel color
```

4 Cooperative BVH Traversal

In this section, we first explain how the baseline BVH traversal works. Then, we propose our cooperative traversal by parallelizing the traversal process.

4.1 Baseline BVH Traversal

The baseline RT unit in Vulkan-sim traverses a BVH using the DFS algorithm, as shown in Algorithm 1 [37].

Algorithm 1: BVH Traversal using DFS to find the closest-hit primitive

```
   Input: ray, root_node
1  if ray intersects root_node then
2      stack.push(root_node);
3  while !stack.empty do
4      node ← stack.pop();
5      if node.type == internal_node then
6          for i = 0 to 5 do // 6-ary tree
7              thit[i] ← intersection_test(ray, node.child[i]);
8              if thit[i] < min_thit then
9                  stack.push(node.child[i]);
10     else // leaf node
11         thit ← intersection_test(ray, node);
12         min_thit ← min(min_thit, thit);
```

The BVH tree in Algorithm 1 is assumed to be a 6-ary tree, meaning each node can have up to 6 child nodes, following the convention used in the MESA graphics library and Vulkan-sim. The nodes in the BVH can be internal nodes, which are nodes that have up to 6 children, or leaf nodes which are primitives like triangles or quads. An internal node contains information such as the coordinates of the AABBs of its child nodes, as well as the address offsets of the child nodes. Leaf nodes contain the vertex coordinates of the primitive.

Tree traversals are initiated when a warp issues a *trace_ray* instruction to the RT unit. Each thread of the warp traverses one ray using the ray properties passed with the *trace_ray* instruction. The root node is an AABB that encompasses the entire scene and is checked for intersection first (line 1). If the root node is hit, its address is pushed onto the traversal stack (line 2). Then, until the stack is empty, the node address at the top of the stack (TOS) is popped, read from memory, and if it is an internal node, its children are checked for intersections and their addresses are pushed onto the stack if hit (lines 4-12). If the popped node is a leaf node, i.e. a primitive, the variable *min_thit*, representing the hit distance of the current closest-hit primitive, is updated if necessary (lines 10-12). An important observation is that an entire tree/sub-tree often does not need to be traversed, as some nodes may be farther than the current closest-hit primitive. Consequently, those nodes and their children can be skipped. Upon finding a primitive hit, threads store the hit information to memory. Typically, subsequent instructions read this information to calculate reflections based on the material and geometry, and trace subsequent bouncing rays.

From the description above, we can see that the baseline traversal is very similar to a generic DFS traversal. Each thread processes the traversal stack one node at a time, i.e., top of the stack, while multiple node addresses are available in the stack. This observation motivates our proposed Cooperative BVH traversal, which makes use of the node addresses in the traversal stack to parallelize and accelerate DFS operations.

4.2 Cooperative BVH Traversal

Our cooperative traversal enables idle (or inactive) threads to be helper threads by tapping into the traversal stacks of busy threads (main threads) within the same warp. We define an *idle* or *helper* thread as one whose traversal stack is empty, and a *busy* or *main* thread as one with a non-empty traversal stack. Algorithm 2 shows how an idle thread behaves during traversal. An idle thread (with its ID as *tid*) first searches for a busy thread within the same warp (lines 2-6). Upon finding a busy thread to help (line 4), the top node is popped from the *main* thread's stack and pushed to the *helper* thread's stack (line 5). The helper thread saves the thread ID of the main thread (line 6), which is used to look-up and update the correct closest hit distance, i.e., *min_thit*. It then proceeds to traverse the tree as usual until its stack is empty (lines 12-20), at which point it will look for another busy thread to help. The traversal finishes when all threads in the same warp have emptied their stacks and updated the destination registers to indicate whether a hit was found, after which the *trace_ray* instruction retires. This cooperative traversal is functionally correct, i.e., the closest-hit primitive will be correctly identified, as long as helper and main threads update the right *min_thit* value whenever a closer hit is found.

Fig. 6 illustrates an example of cooperative traversal. In the baseline, the entire tree is traversed by a single thread. After checking the AABB of root node for intersection and finding a hit, its address is pushed onto the stack to initiate the traversal. Then, the root node address is popped and fetched from memory. When the node data arrives, the children are tested for intersection, and found that they are both hit, therefore both the child node addresses are pushed onto the stack. Next, the thread pops the left child address and starts traversing the left subtree before checking the right subtree. With cooperative traversal, let us assume there is one helper thread. After the main thread pops the root's left child address, the helper thread pops the main thread's stack and gets the root's right child address. Therefore, the helper thread would traverse the right subtree of the root. With both the main and the helper threads, the two subtrees are traversed in parallel. Note that it is possible that the helper thread pops a different node address from the main thread, depending on when the helper thread is available to pop an address from the main thread's traversal stack. In this case, different subtrees would be traversed in parallel. Whenever a thread empties its traversal stack, it would become a helper thread and try to take an address from a busy thread's stack. As such, the degree of parallelization is not affected by which address is taken by a helper thread. Although both the main and helper threads find triangle hits (red circles in Fig. 6), only one of them is identified as

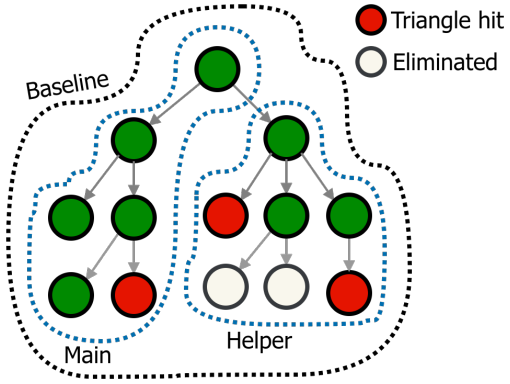


Figure 6: Example BVH tree traversal comparing baseline and cooperative traversal.

the closest-hit primitive, as the threads compare with the current closest-hit, *min_thit*, of the main thread before updating it. As a result, the correctness of traversal is maintained.

While we focus on cooperative traversal for DFS, it can be extended to breadth-first-search (BFS) as BFS is also inherently parallelizable. Compared to DFS, BFS would use a queue (FIFO) rather than a stack (LIFO) to track nodes. In that case, helper threads would steal nodes from the front of the queue and start their traversal. In general, as long as a tree/graph traversal algorithm is parallelizable and uses a stack/queue to track nodes, cooperative traversal can be directly applied.

An important design consideration in cooperative traversal is deciding the range of the threads who can help each other. In Algorithm 2, all 32 threads in a warp are allowed to help each other, which maximizes cooperation and performance, but at the cost of more complex hardware. We investigate more restrictive configurations where only threads within the same *subwarp* are allowed to help each other, in order to reduce hardware overhead. We explore area and performance impact of subwarp sizes of 4, 8 or 16 threads in Section 7.

Algorithm 2: Cooperative BVH Traversal to find the closest-hit primitive

```
Input: rays, min_thits, stacks, mtids, root_node
1  mtid[tid] ← tid // Initialize mtid to thread id
2  if stacks[tid].empty then
3      for i ← 0 to warp_size do
4          if !stacks[i].empty then
5              stacks[tid].push(stacks[i].pop());
6              mtid[tid] ← mtids[i] // Save main thread id to
                 mtid
7              break;
8  stack ≡ stacks[tid] // Let stack refer to stacks[tid]
9  mtid ≡ mtid[tid] // Let mtid refer to mtid[tid]
10 ray ≡ rays[mtid] // Let ray refer to rays[mtid]
11 min_thit ≡ min_thits[mtid]
12 while !stack.empty do
13     node ← stack.pop();
14     if node.type == internal_node then
15         for i ← 0 to 5 do
16             thit[i] ← intersection_test(ray, node.child[i]);
17             if thit[i] < min_thit then
18                 stack.push(node.child[i]);
19     else
20         thit ← intersection_test(ray, triangle);
21         min_thit ← min(min_thit, thit);
```

5 CoopRT Architecture

5.1 Overview of the Architecture

To support our cooperative BVH traversal, we modify the warp buffer and the accompanying logic in the RT unit. Fig. 7 shows the high level block diagram of our proposed implementation, with the added per-thread structures highlighted using red and added per-RT unit structures highlighted using purple. At every cycle, the warp scheduler in the RT unit picks a non-stalling warp from the warp buffers ❶. The memory scheduler iterates through the threads in the scheduled warp, checking each thread's status to determine if it has any remaining memory requests. The node addresses from the TOSes of these threads are coalesced to remove redundant cache or memory accesses. One of these unique addresses is selected and added to the memory access queue, which breaks the requests into small chunks before sending them to memory hierarchy ❷. The threads that generated this request pop their TOSes and save the popped addresses in registers, which will be needed when the memory responses come in. In parallel, the Load Balancing Unit (LBU) looks for a thread that needs help, and another thread that can offer help within the scheduled warp ❸. If a main and a helper thread are picked, it pushes the node at the TOS of the main thread to helper thread's stack by controlling the per-thread multiplexors ❹. If the Helper ID that LBU finds matches the thread's ID, then the multiplexors select the TOS coming from LBU, which is then pushed to this thread's traversal stack. Since LBU moves only one node at a cycle, the number of pushes is set to 1; whereas the math units may push up to 6 nodes at once depending on how many child nodes are hit.

Responses from memory hierarchy are inserted to the Response FIFO, and fed into the Math units where coordinate transformations, ray-box and ray-triangle intersection tests are carried out ❺. The address of the response is checked against the saved TOS registers to determine which thread or threads originated the request. We assume there is one math unit associated with each thread to ensure there are no stalls for intersection tests, similar to [37]. Depending on the intersection test results, child nodes are pushed to traversal stacks of the associated threads. If a primitive hit is found, the closest hit information, *min_thit*, may be updated (Section 5.3) and a store request for the primitive data is inserted to the store queue which can then be read by the closest-hit or any-hit shaders [37].

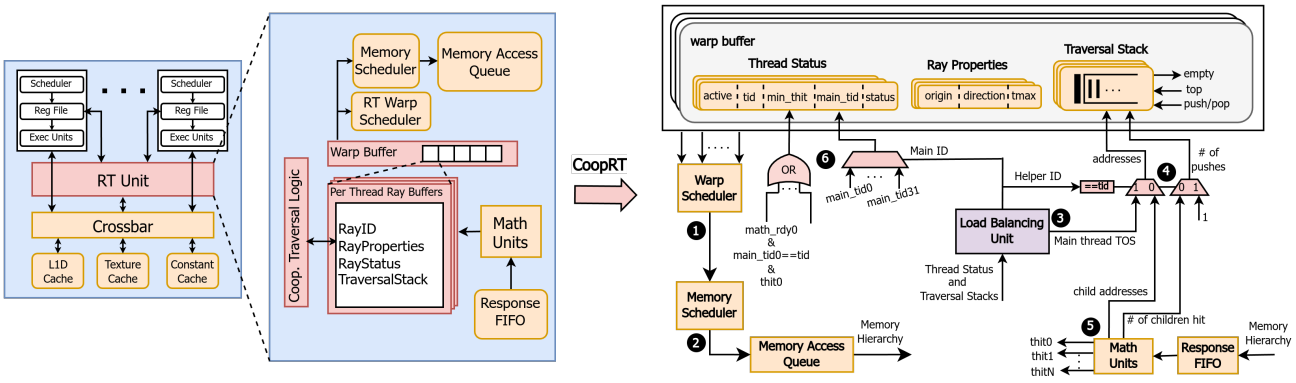


Figure 7: Overview of the modified RT unit. Orange blocks indicate existing hardware, red blocks are newly added per-thread structures, and purple blocks are newly added per-RT unit (therefore per-SM) structures. *main_tid* is a new 5-bit field added to Thread Status in the warp buffer.

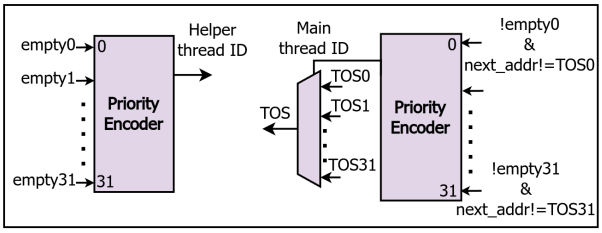


Figure 8: Load Balancing Unit.

5.2 Load Balancing Unit

LBU is a per SM unit and is responsible for assigning idle threads as helpers to a busy thread, and moving node addresses from the main to helper threads. Fig. 8 shows details of the LBU. The priority encoder (PE) on the right of Fig. 8 determines which thread needs help, and outputs its thread ID. A thread needs help if its traversal stack is not empty, and its TOS is not being processed in that cycle. Both active and inactive threads in the warp can be helped, which means a helper thread can also be helped by another thread as long as its traversal stack is not empty. The main thread ID is used to control the multiplexor which outputs the TOS of main thread. The PE on the left determines which thread is available to help, and outputs its thread ID. The empty signal means the corresponding thread's traversal stack is empty at the cycle. If both a main and a helper thread are found, *main_tid* of the main thread is saved in the helper thread's *main_tid* field in the warp buffer using the per-thread multiplexor ❻. All the threads traversing the same ray, including the helper(s) and main, use the *main_tid* field to get the right ray properties and *min_thit* value. *main_tid* is initialized to *tid* when the *trace_ray* instruction first enters the RT unit. As threads are assigned as helpers, they save the main thread's *main_tid* field in their own *main_tid* field.

5.3 Synchronization Between Main and Helper Threads

When multiple threads traverse the same ray, to ensure functional correctness, they need to update the same *min_thit* register as they find primitive hits. The *min_thit* field in the warp buffer stores the hit distance of the current closest-hit primitive, and it is updated only when a closer primitive is found. With CoopRT, all helper threads update the *min_thit* field of the main thread, which ensures functional correctness. This is achieved via the logic shown in Fig. 7, which is a per-thread structure ❻. Three signals are ANDed together for each thread: *math_rdy* signal that indicates the math unit's output is ready, the *main_tid==tid* signal where *main_tid* is the main thread's ID saved by the helper thread and *tid* is the thread ID of the thread whose *min_thit* is being updated, and the *thit* value itself. Output of all the AND gates (one AND per thread) are ORed to allow the valid *thit* value that will be written to the main thread's *min_thit* field if it is smaller than the current one. The OR gate behaves like a multiplexor, because it is logically impossible for more than one thread to find a primitive hit for a given ray at the same cycle. The reason is that the responses from the response FIFO are popped one per cycle, and the math unit latency is constant. If the bandwidth of the response FIFO is increased to be more than one response per cycle, we can let each helper update their own min_thit field first and then borrow atomic instruction support (e.g., atomicMin) to update the main threads' *min_thit* at the store buffer when retiring. Since the FIFO throughput is not a performance bottleneck, we do not investigate this design option further.

It is possible that a main thread empties its traversal stack, i.e. finishes traversal, before a helper thread. This does not pose a problem as the *trace_ray* instruction will not retire until all threads in the warp have emptied their stacks.

Overall, the newly added support shown in Fig. 7 provides (a) a mechanism to identify helper and main thread pairs, up to one pair a cycle by the LBU, (b) the data path to read from the main thread's traversal stack and write to the helper thread's traversal stack (a bus design can also be sufficient, as only one main-helper pair is selected per cycle), and (c) the data path from the math units' outputs, i.e., the *thit* from a helper thread, to update the

main thread's *min_thit* if it is smaller than the *min_thit*. This is achieved with a crossbar. If we allow any thread in a warp to help each other, it is a 32x32 crossbar. If we limit the scope to a subwarp (e.g., size 8), then we can replace the 32x32 crossbar with multiple small crossbars (e.g., 4 8x8 crossbars). The logic is simplified by the observation that only up to one helper thread would update the main thread's *min_thit* at a cycle. The reason is that node addresses are unique, the memory response IO throughput is one per cycle, and different helper threads traverse different portions of the BVH tree. As a result, it is impossible for more than one thread to ever access the same primitive for a given ray, and when different threads access different primitives, their *thit* values will be available at different cycles.

6 Experimental Methodology

6.1 Modeling CoopRT in Vulkan-sim

We modify Vulkan-sim 2.0 [37] to model CoopRT and evaluate its performance. Vulkan-sim is built on top of GPGPUsim [31], which is made up of a functional and a timing simulator. Vulkan-sim performs the actual BVH traversal process in the functional simulator, and passes a list of BVH node addresses for each thread to the timing simulator, which in turn simulates the memory accesses. The timing simulator picks a warp from the warp buffer at every cycle, reads the node addresses from the top of the lists passed from the functional simulator, merges duplicate addresses and sends one of the unique addresses to memory hierarchy. To implement CoopRT, we check for idle threads in the scheduled warp each cycle. If any are found, the node at the top of the list of a busy thread is moved to the idle thread's node list.

The functional simulator assumes a single thread traverses the BVH tree in DFS fashion for a given ray, and therefore generates the list of nodes accordingly. This means some nodes in the tree get eliminated during traversal because they are farther than the current closest-hit primitive, and they are not added to the list. However, when multiple threads traverse the BVH together, it is impossible to know beforehand which nodes will be eliminated, because that depends on runtime information such as how many threads traverse the tree and which parts of it they are processing, which is not available in the functional simulator.

We resolve this issue by not doing any node eliminations in the functional simulator, and instead, passing the *thit* values of each node to the timing simulator. In the timing simulator, we keep track of the *min_thit* value for each thread, which is initialized to positive infinity. Before processing a new node in the list, we compare its *thit* value to the *min_thit* value. If *thit* is greater than or equal to *min_thit*, we discard it.

To estimate the impact on power consumption, we use GpuWattch [33] shipped with Vulkan-sim.

We use one of the default configurations available in the Vulkan-sim repository, namely the *SM75_RTX2060* configuration. Table 1 shows the key settings in this configuration.

# Streaming Multiprocessors(SM)	30
Max. TBs per SM	32
Warp Size	32
Instruction Cache	128KB, 20 cycles
L1 Data Cache	64KB, Fully assoc. LRU, 20 cycles
L2 Cache	3MB, 16-way assoc. LRU, 160 cycles
Core, Interconnect, L2 Clock	1365 MHz
Memory Clock	3500 MHz
# of RT Units per SM	1
RT Unit Warp Buffer Size	4

Table 1: Vulkan-sim baseline hardware configuration.

6.2 Benchmark Suite

We use the Lumibench[35] ray tracing benchmark suite for evaluation. Lumibench features 16 3D scenes with various geometric complexities and lighting conditions. Table 2 shows a summary of the 3D scenes.

The highest resolution we could simulate without simulations timing out or running out of memory is 256x256. Among all the scenes, we could only simulate 13 scenes at this resolution. For the remaining ones, we run the scenes **car** and **robot** at the resolution of 128x128 because they either time out or consume too much memory at the resolution of 256x256. The scene **park** would not finish after 3 days of simulation and time out at the resolution of 128x128. In all the scenes, we use 1-sample-per-pixel, meaning one primary ray for each pixel.

At the resolution of 256x256, there are 2048 thread blocks (TB) and each TB has one warp, which is the default thread block size in Vulkan-sim. 2048 TBs are enough to fill up the entire GPU that we use in this study, which has 30 SMs.

7 Results

7.1 CoopRT Performance and Memory Bandwidth Utilization

Fig. 9 shows the normalized speedup and power of CoopRT with PT shaders. We observe up to 5.11x speedup, with a geometric mean of 2.15x. Scenes with low SIMT efficiency and long BVH traversals, such as **crnvl**, **fox** and **party**, benefit the most from cooperative traversal. Although **spnza** has the highest number of BVH nodes visited among all the scenes, it also has relatively higher SIMT efficiency, meaning there are fewer idle threads, likely because it is a closed scene with minimal exposed sky. On average, power consumption is increased by 2.02x, and energy is decreased to 0.94x.

To provide more insight to the source of performance gains, we calculate average thread utilization in RT unit using the readily available AerialVision stats [10]. At every 500 GPU cycles, we collect the number of busy threads in RT unit, i.e., the threads with non-empty traversal stacks, and divide them by the number of total threads to get the thread utilization per sample. We then average all the samples to obtain overall utilization for each scene. Fig. 10 shows the overall thread utilization for baseline and CoopRT.

From the Figs. 9 and 10, we can see that the speedups are proportional to the thread utilization *improvements*, rather than the actual final utilization. The three scenes, **crnvl**, **fox**, and **party** have the highest *improvement* in utilization, which is why they achieve the highest speedups. In other words, CoopRT overcomes the divergent nature of ray tracing by exploiting the parallelism of BVH traversal. The more divergent a scene, the higher *speedup* is achieved.

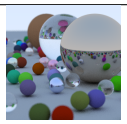
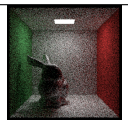
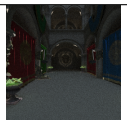
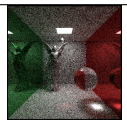
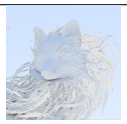
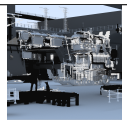
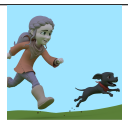
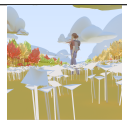
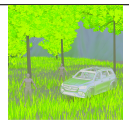
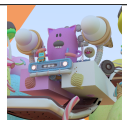
Scene								
Label	**wknd**	**ship**	**bunny**	**spnza**	**chsnt**	**bath**	**ref**	**crnvl**
Tree Size(MB)	0.2	0.5	12.2	22	25.5	104.2	37.1	37.3
Depth	7	12	11	16	12	16	13	16
Scene								
Label	**fox**	**party**	**sprng**	**lands**	**frst**	**park**	**car**	**robot**
Tree Size(MB)	597.8	143.8	164.3	279.2	348.6	501.9	1,233.6	1,721.3
Depth	15	14	14	12	14	14	16	18

Table 2: Benchmark scenes from LumiBench [35]. Scene stats taken from [15].

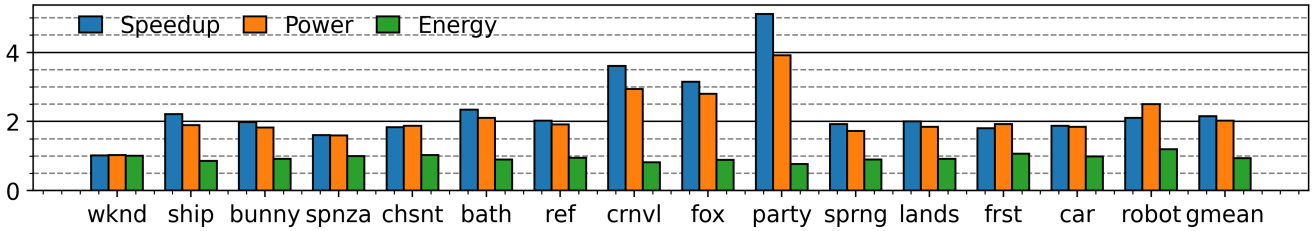


Figure 9: Normalized speedup, power and energy consumptions of CoopRT over the baseline RT unit.

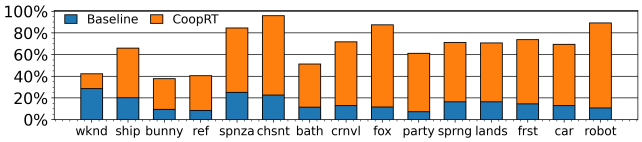
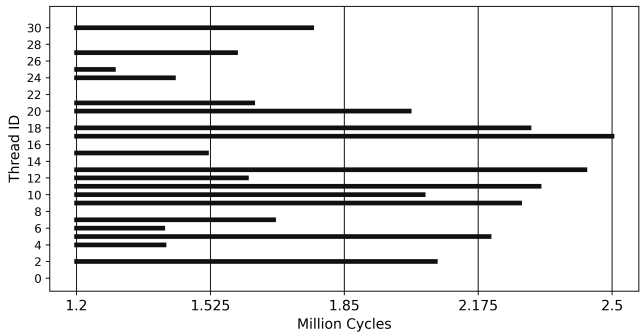


Figure 10: Average thread utilization.

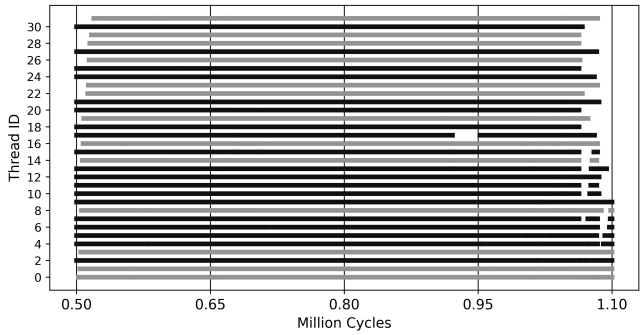
Fig. 11 presents the *trace_ray* instruction execution timeline of an example warp (in the **bath** scene) to illustrate how the thread utilization increases with CoopRT. In Fig. 11a, there are 13 inactive threads, and several threads that finish their traversal early and idle, yielding an average utilization rate of 30.5%. Fig. 11b shows how the timeline changes with CoopRT. Inactive threads steal work from active threads, and spend most of their time doing traversals. In addition, active threads who finish early also steal work from other threads. Ultimately, average utilization increases to 94.6%.

CoopRT also entails substantial improvement in memory bandwidth utilization due to the increased number of threads doing traversals in parallel. Fig. 12 shows the L2 cache and DRAM bandwidth utilization normalized to baseline. We observe up to 5.7x and 5.5x increase in L2 and DRAM bandwidth respectively. This increase is primarily due to low bandwidth use of the baseline RT unit, as there are fewer busy threads in the baseline.

Another important factor is the number of maximum number of warps allowed in the RT unit. By default, as shown in Table 1, the RTX2060 configuration allows at most 4 warps to exist simultaneously in the RT unit. However, 4 warps are not enough to fully utilize the memory bandwidth. Simply increasing the number of warp buffers in the RT unit is costly, as all the fields in the warp buffer add up to hundreds of bits of storage per thread. We discuss the area overhead with detail in Section 7.5. To evaluate the impact of warp buffer size on performance, we simulate different warp buffer sizes with and without CoopRT. Fig. 13 shows the normalized speedups for different warp buffer sizes. Compared to the baseline, we see geometric means of 1.45x, 1.64x, 1.64x for warp buffer sizes of 8, 16, 32 without CoopRT. As the warp buffer size increases, inter-warp parallelism and memory bandwidth utilization increase, yielding higher throughput and performance. Increasing the warp buffer size from 4 to 8 provides the greatest performance boost, with further increases yielding diminishing returns. For this particular hardware configuration, 8 or 16 buffer entries seem like the sweet spot for performance and area trade off. When CoopRT is enabled, the impact of warp buffer size becomes less significant. We see geometric means of 2.15x, 2.13x, 2.06x, 1.99x for warp buffer sizes of 4, 8, 16 and 32 over the baseline. This is because CoopRT already saturates the memory bandwidth utilization. CoopRT with just 4 warp buffer entries achieves greater speedup than the baseline 32 entry warp buffer configuration. Moreover, CoopRT reduces the latency of the longest-running or slowest warps, which would determine the frame rate in real-time rendering, compared to the schemes using large warp buffers. Fig. 14 shows the latency of the longest running warp in each scene normalized to baseline. From the figure, we can see that CoopRT achieves higher throughput and better latency via intra-warp parallelism. On average, CoopRT achieves 0.46x the latency of the baseline, compared to 0.62x achieved by the large warp buffer scheme.



(a) Baseline



(b) CoopRT

Figure 11: RT unit *trace_ray* instruction execution timelines. Dark and light gray bars represent active and originally inactive threads, respectively. A continuous bar indicates a non-empty traversal stack. bath scene, 256x256 resolution path tracing.

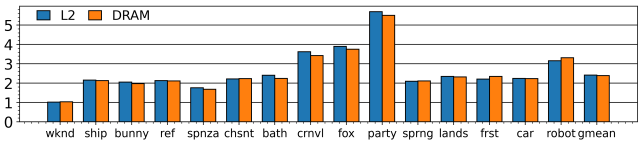


Figure 12: Normalized L2 ↔ Interconnect, and DRAM bandwidth with CoopRT over the baseline.

To quantify and compare the energy efficiency of CoopRT against large warp buffers, we calculate the energy-delay products (EDP) [24][12] of each approach, and plot the EDP improvements in Fig. 15. Geometric means are 1.54x, 1.75x, 1.75x and 2.29x for warp buffer sizes of 8, 16, 32 without CoopRT and 4 with CoopRT respectively. While neither approach introduces substantial energy consumption, CoopRT with the warp buffer size of 4 achieves higher performance by better utilizing the existing hardware, thus achieving better EDP with much smaller area overhead.

7.2 Memory Contention Under CoopRT

When CoopRT is enabled, the RT unit generates more memory requests in a shorter amount of time. This could lead to contention in the memory hierarchy. To analyze such contention, we collect the L1 and L2 miss rates for each scene, as shown in Fig. 16. From the figure, we observe that CoopRT results in: (1) increased L1 cache miss rates indicating more contention on L1; (2) a higher number of L2 accesses but similar L2 miss rates meaning more reuses at the L2 (as some of the original L1 reuses now happen at L2); (3) GPU latency hiding capability tolerating additional L1 misses; and (4) overlapping misses (i.e. memory level parallelism) and memory bandwidth utilization being more important than the number of misses alone.

7.3 Ambient Occlusion and Shadow Shaders

In addition to PT shaders, Lumibench features ambient occlusion (AO) and shadow (SH) shaders that use the ray tracing pipeline to produce realistic lighting effects. Unlike PT, these shaders are typically used together with rasterization, and are readily employed in real-time applications such as video games. AO and SH shaders are much more lightweight than PT, as they do not aim to render the entire 3D scene, but rather perform lighting calculations. Similar to PT, AO and SH shaders also begin with primary rays generated from the camera, but instead of bouncing the primary ray through the scene, they find the closest object that the primary rays hit. Then, from that intersection point, they trace a small number of shadow rays to determine how much light reaches that point. The key difference from PT is that these shadow rays are much more localized and coherent (i.e., non-divergent). Therefore, they are relatively faster to trace and there is less room for improvement. Fig. 17 shows how CoopRT performs with AO and SH shaders. As expected, the speedups are smaller compared to PT. This is because, as mentioned before, AO and SH rays do not diverge nearly as much as PT, leaving less speedup opportunity for CoopRT than PT rays. Despite this, CoopRT achieves an average of 1.42x and 1.28x speedup for AO and SH, respectively.

7.4 Mobile GPU Configuration

To demonstrate the robustness of CoopRT, we also evaluate the performance on a mobile GPU configuration included in Vulkan-sim, which has 8 SMs and 4 memory channels. Fig. 18 shows the speedup, power and energy results under this configuration. We see CoopRT achieves an average of 1.8x speedup, 1.71x power and 0.95x energy relative to the baseline. The speedups are mainly bottlenecked by the memory bandwidth limitation of this configuration. When CoopRT is enabled, DRAM utilization increases from 44.0% to 85.3% on average.

7.5 Area Overhead

We implement the hardware model proposed in Section 5 to estimate the area cost of CoopRT. We wrote the RTL for all of the newly introduced blocks, and synthesized using FreePDK45 [38] and Synopsys Design Compiler. While the proposed logic is purely combinational that consists of PEs, multiplexors and logic gates, we also take into account the extra fields introduced to warp buffers.

The total number of combinational cells in this design is 16,122, occupying an area of 13,347 μm^2. For reference, each sequential cell (e.g., a D flip-flop) takes up 6 μm^2 in this design kit. This means that the area occupied by the combinational logic is equivalent to approximately 2,200 flip-flops. By comparison, in the baseline RT unit, just the *RayProperties*, *TraversalStack* and *min_thit* fields

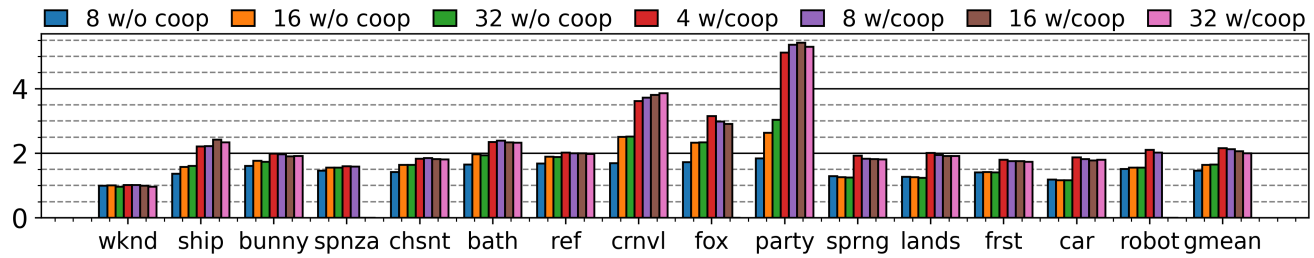


Figure 13: Normalized speedups for different RT warp buffer sizes with and w/o CoopRT. Baseline is 4-entry warp buffer without CoopRT. 1 warp per thread block, 32 thread blocks in one SM at a time. Missing data points are due to consistently crashing or timing out simulations.

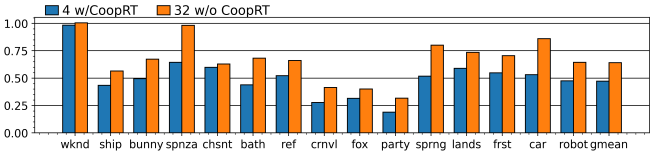


Figure 14: Latency of the slowest warps, normalized to baseline (4 buffer entries without CoopRT). Lower the better.

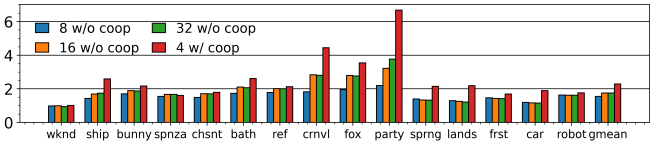


Figure 15: Normalized improvement in EDP for different RT warp buffer sizes. Baseline is 4-entry warp buffer without CoopRT.

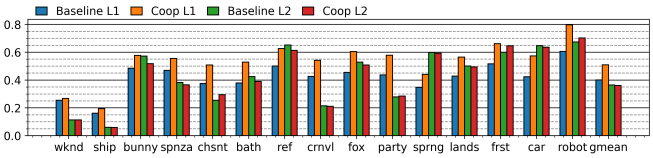


Figure 16: L1 and L2 cache miss rates.

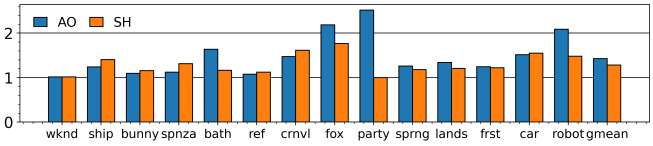


Figure 17: Speedups of CoopRT for AO and SH shaders normalized to baseline.

in a warp buffer require a total of 768 bits of storage per thread, assuming a 16-entry traversal stack. Here, *RayProperties* includes the ray's origin, direction, and a *max_thit* value. The extra fields in the warp buffer include the 5-bit *main_tid* field per thread, and a stack empty flag per thread. Assuming 4 warp buffer entries, and 32 threads per warp, the warp buffer takes up 98,304 (=4*32*768) bits of storage.

This means the CoopRT hardware takes up less than 3.0% ((2200+4*32*(5+1))/98304) of the warp buffer area. This comparison also shows that CoopRT is much more area efficient than simply

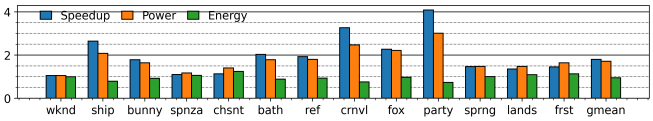


Figure 18: Speed, power and energy of CoopRT on a mobile GPU, normalized to baseline.

increasing the number of warp buffers as each warp buffer entry takes 24,576 (=32*768) bits.

One way to reduce the area consumption is to implement a *subwarp* scheme where only the threads within the same subwarp are allowed to help each other. A subwarp is a smaller, fixed-size group of threads in a warp. This slightly reduces the hardware cost, as the per-thread structures shown in Fig. 7 do not need to have 32 inputs coming from 32 threads. The smaller the subwarp size, the less area consumed. However, this also places a constraint on which threads can help each other, therefore reducing the amount of parallelism and performance.

One important design decision to make with the subwarp scheme is if all subwarps are processed together in one cycle to find a pair of helper-main threads for each subwarp, or if just one subwarp is picked and processed every cycle. The former approach reduces area consumption by reducing the number and sizes of the per-thread OR gate and multiplexor in Fig. 7. It also replaces the PEs with a smaller pair for each subwarp. The latter approach rescales the number of gates and multiplexors to the subwarp size, potentially eliminating significant amount of hardware. However, it also requires additional hardware for subwarp scheduling. At each cycle, the subwarp scheduler would have to look through the subwarps, and pick a suitable one that has a main-helper pair candidate. In terms of performance, both approaches would perform similarly, as the latency of a *trace_ray* instruction is on the order of thousands of cycles, which is long enough to hide any subwarp scheduling latency.

Using the first approach described before, we synthesize CoopRT hardware with subwarp sizes of 4, 8 and 16 to estimate area savings, and also simulate in Vulkan-sim to understand the performance trade off. Table 3 shows the area results for different subwarp sizes. By decreasing the subwarp size to 4, almost 10% of the area can be saved. Fig. 19 shows how the performance changes with subwarp size. As expected, reducing the subwarp size leads to a decline in performance. Average speedups are 1.72x, 1.97x, 2.09x and 2.15x for 4, 8, 16 and 32 respectively. It is worth noting that both area

Subwarp size	# of cells	Total area(μm²)	Percent change(Area)
32	16122	13347	0
16	15867	13104	1.8
8	15511	12661	5.1
4	15167	12055	9.7

Table 3: Area results for different subwarp configurations. Percent change is relative to subwarp size 32.

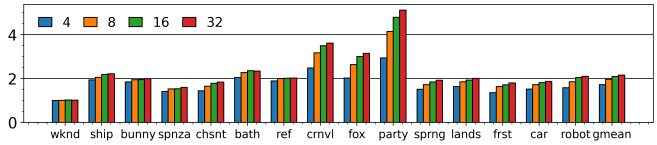


Figure 19: Speedups of CoopRT for subwarp sizes of 4, 8, 16 and 32 normalized to baseline.

consumption and performance drop the most when changing the subwarp size from 8 to 4.

8 Related Work

8.1 Ray Tracing on GPUs

Due to its parallel nature, ray tracing has been implemented and studied on GPUs. Early research on GPU ray tracing utilized GPGPU programming models such as CUDA, due to the lack of hardware and API support. Aila et al. [9] implement a GPU ray tracer to assess the performance and bottlenecks of ray traversal on GPUs. They explore replacing early terminated rays with new ones, wider BVH trees, and work queues to improve SIMD efficiency. In another work [8], Aila et al. focus on incoherent (divergent) rays and explore a treelet based BVH traversal scheme. The BVH tree is statically split into smaller trees called treelets to shrink the working set and reduce the memory footprint and latency. Wald [42] proposes active thread compaction to mitigate divergence in PT. At the beginning of each ray bounce, active threads across multiple warps are compacted together to form fewer but more efficient warps, which is similar to the idea in [21]. Therefore, it may address the inactive thread problem to some degree (as it needs to compact different numbers of TBs or warps for each bounce), but not early finishing threads.

8.2 Hardware Accelerated Ray Tracing

The recent introduction of specialized RT units in commodity GPUs sparked architectural research for hardware acceleration of ray tracing. Lufei et al.[34] propose an intersection prediction algorithm. A dedicated hardware cache is used to store the intersection results of previous rays. Future rays can look up the cache by calculating a hash using the ray properties and predict intersections without traversing the BVH tree. Although effective with localized rays that AO and SH shaders generate, its effectiveness with PT is unknown. Saed et al. [37] extend GPGPUsim 4.0 [31] by incorporating an RT unit which is capable of simulating Vulkan ray tracing shaders. Inspired by Aila [8], Chou et al. [15] propose a treelet based BVH traversal and hardware prefetcher for ray tracing. Prefetching is a viable solution for memory latency-bound workloads, such as PT. In this context, the treelet prefetcher proves to be useful, though it requires complex hardware, and a custom BVH organization. CoopRT can be combined with a prefetcher, such as the Treelet prefetcher, although the benefits would need more careful consideration. The reason is that CoopRT increases parallelism and may saturate the memory bandwidth. In this case, the bandwidth left for prefetching would be limited. In a system where bandwidth is abundant, CoopRT can benefit from prefetching due to reduced memory access latency.

9 Conclusion

In this work, we propose a novel cooperative BVH traversal scheme, CoopRT, to accelerate GPU ray tracing. We capitalize on two important insights: low SIMT efficiency of ray tracing workloads, and the inherent parallelism of BVH traversal.

Ray tracing applications traverse millions of rays per frame, and during the traversal, a large number of rays terminate early either because they miss the scene or hit a light source. This causes the threads to idle, while other threads continue their traversals. However, BVH traversal can be parallelized without losing functional correctness. Therefore, we propose to utilize the idle threads to help the busy threads finish their traversals faster by letting the idle threads steal BVH nodes from busy threads' traversal stacks. We evaluate the performance of CoopRT in Vulkan-sim [37] by extending the baseline RT unit. We simulate CoopRT across 13 scenes in Lumibench [35], and show that CoopRT achieves up to 5.11x speedup, with an average of 2.15x compared to the baseline RT unit. We also propose a hardware model for CoopRT and implement it in RTL to estimate its area overhead. We show that CoopRT takes less than 3.0% of the warp buffer area in the RT unit.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. The work is funded in part by NSF grants PHY-2325080 (with a subcontract to NC State University from Duke University), and OMA-2120757 (with a subcontract to NC State University from the University of Maryland).

A Artifact Appendix

A.1 Abstract

This artifact provides the source code for the modified Vulkan-sim which has newly added configuration options to enable CoopRT. In addition, Python and shell scripts are provided to run all the necessary simulations and generate the plots and figures in this paper. To streamline the artifact evaluation process, we prepared a docker image with all the software dependencies installed. As simulations take a long time, we also included the raw simulation outputs that we generated and used for this paper.

A.2 Artifact check-list (meta-information)

- **Program:** Vulkan-sim, RayTracingInVulkan
- **Compilation:** gcc/g++, ninja, meson, cmake, nvcc
- **Run-time environment:** Ubuntu 20.04
- **Hardware:** 32+GB RAM
- **Metrics:** Number of cycles, average power consumption
- **Output:** Vulkan-sim simulation outputs, figures.

- **How much disk space required (approximately)?:** 30GB
- **How much time is needed to prepare workflow (approximately)?:** About 1 hour
- **How much time is needed to complete experiments (approximately)?:** 5 minutes to generate figures. One week for Vulkan-sim simulations, if ran in parallel.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Yes
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.15103378

A.3 Description

A.3.1 How to access. We provide the Docker image which has everything required to run the simulations and generate the figures. You can download it from Zenodo using the archived link.

A.3.2 Hardware dependencies. Only requirement is 32+GB of memory.

A.3.3 Software dependencies. Only a Docker installation is required. All software dependencies are installed in the Docker image.

A.4 Installation

Download the docker image from Zenodo, and start a container using the commands below,

```
docker load < cooprt-isca2025-ae.tar.gz
docker run -it cooprt-isca2025-ae:1.0 /bin/bash
```

A.5 Experiment workflow

Inside the container, we provide a shell script `cooprt.sh` that has all the simulation commands needed to generate results. However, due to the large number of simulations, we do not recommend running the shell script directly, as it runs the simulations sequentially. Instead, depending on the resources, simulations should be run in parallel. The shell script simply serves as a reference for simulation commands. Workflow for launching parallel jobs depends on the system being used, therefore we cannot provide a one-for-all script to launch parallel simulations. To launch a simulation in the container,

```
cd /home/root/vulkan-sim-root
source embree-3.13.4.x86_64.linux/embree-vars.sh
source vulkan-sim/setup_environment
cd RayTracingInVulkan/build/linux/bin
./RayTracer --scene 20 --width 256 \
--height 256 > ship_pt.log
```

We also provide all of the raw simulation logs and the Python scripts that we used to plot the figures in this paper. To generate the figures, following command can be used inside the container,

```
python3 figure1.py
```

This will generate `fig1.png` using the simulation logs under `cooprt_raw_simulation_results`. Other figures can be generated in a similar fashion.

A.6 Evaluation and expected results

Running the Python scripts will generate the figures using the existing simulation logs. To reproduce or replace the simulation logs, the simulation commands in the shell script can be used.

References

[1] [n. d.]. Cyberpunk 2077: Technology Preview Of New Ray Tracing Overdrive Mode Out Now. https://www.nvidia.com/en-us/geforce/news/cyberpunk-2077-ray-tracing-overdrive-update-launches-april-11/

[2] [n. d.]. Intel Embree. https://www.embree.org/

[3] [n. d.]. Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in... https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html

[4] [n. d.]. NVIDIA ADA GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[5] [n. d.]. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[6] [n. d.]. Real-Time Ray Tracing. https://dev.epicgames.com/documentation/en-us/unreal-engine/hardware-ray-tracing-tips-and-tricks-in-unreal-engine

[7] [n. d.]. Real-time Raytracing for Interactive Global Illumination Workflows in Frostbite. https://www.gdcvault.com/play/1024801/

[8] Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics (HPG '10)*. Eurographics Association, Goslar, DEU, 113–122.

[9] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. Association for Computing Machinery, New York, NY, USA, 145–149. https://doi.org/10.1145/1572769.1572792

[10] Aaron Ariel, Wilson W. L. Fung, Andrew E. Turner, and Tor M. Aamodt. 2010. Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 164–174. https://doi.org/10.1109/ISPASS.2010.5452029

[11] Aaron Barnes, Fangjia Shen, and Timothy G. Rogers. 2024. Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*. Association for Computing Machinery, New York, NY, USA.

[12] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. 2000. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro* 20, 6 (Nov. 2000), 26–44. https://doi.org/10.1109/40.888701 Conference Name: IEEE Micro.

[13] Brian Caulfield. 2018. What's the Difference Between Ray Tracing and Rasterization? https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/

[14] Brian Caulfield. 2022. What Is Path Tracing? https://blogs.nvidia.com/blog/what-is-path-tracing/

[15] Yuan Hsi Chou, Tyler Nowicki, and Tor M. Aamodt. 2023. Treelet Prefetching For Ray Tracing. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 742–755.

[16] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*. 1–6. https://doi.org/10.1109/RT.2006.280208

[17] Per H. Christensen and Wojciech Jarosz. 2016. The Path to Path-Traced Movies. *Foundations and Trends® in Computer Graphics and Vision* 10, 2 (2016), 103–175. https://doi.org/10.1561/0600000073

[18] Sana Damani, Mark Stephenson, Ram Rangan, Daniel Johnson, Rishkul Kulkami, and Stephen W. Keckler. 2022. GPU Subwarp Interleaving. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1184–1197. https://doi.org/10.1109/HPCA53966.2022.00090 ISSN: 2378-203X.

[19] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. 2014. A scalable multi-path microarchitecture for efficient GPU control flow. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 248–259. https://doi.org/10.1109/HPCA.2014.6835936 ISSN: 2378-203X.

[20] Robert Felbecker, Leszek Raschkowski, Wilhelm Keusgen, and Michael Peter. 2012. Electromagnetic wave propagation in the millimeter wave band using the NVIDIA OptiX GPU ray tracing engine. In *2012 6th European Conference on Antennas and Propagation (EUCAP)*. 488–492. https://doi.org/10.1109/EuCAP.2012.6206198 ISSN: 2164-3342.

[21] Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, USA, 25–36.

[22] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, USA, 407–420. https://doi.org/10.1109/MICRO.2007.12

[23] Jeffrey Goldsmith and John Salmon. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications* 7, 5 (May 1987), 14–20. https://doi.org/10.1109/MCG.1987.276983 Conference Name: IEEE Computer Graphics and Applications.

[24] R. Gonzalez and M. Horowitz. 1996. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits* 31, 9 (Sept. 1996), 1277–1284. https://doi.org/10.1109/4.535411 Conference Name: IEEE Journal of Solid-State Circuits.

[25] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *2007 IEEE Symposium on Interactive Ray Tracing*. 113–118. https://doi.org/10.1109/RT.2007.4342598

[26] Dongho Ha, Lufei Liu, Yuan Hsi Chou, Seokjin Go, Won Woo Ro, Hung-Wei Tseng, and Tor M. Aamodt. 2024. Generalizing Ray Tracing Accelerators for Tree Traversals on GPUs. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*. Association for Computing Machinery, New York, NY, USA.

[27] Danping He, Bo Ai, Ke Guan, Longhe Wang, Zhangdui Zhong, and Thomas Kürner. 2019. The Design and Applications of High-Performance Ray-Tracing Simulation Platform for 5G and Beyond Wireless Communications: A Tutorial. *IEEE Communications Surveys & Tutorials* 21, 1 (2019), 10–27. https://doi.org/10.1109/COMST.2018.2865724 Conference Name: IEEE Communications Surveys & Tutorials.

[28] Jakob Hoydis, Faycal Ait Aoudia, Sebastian Cammerer, Merlin Nimier-David, Nikolaus Binder, Guillermo Marcus, and Alexander Keller. 2023. Sionna RT: Differentiable Ray Tracing for Radio Propagation Modeling. In *2023 IEEE Globecom Workshops (GC Wkshps)*. 317–321. https://doi.org/10.1109/GCWkshps58843.2023.10465179

[29] Thiago Ize, Ingo Wald, and Steven G. Parker. 2007. Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the 7th Eurographics conference on Parallel Graphics and Visualization (EGPGV '07)*. Eurographics Association, Goslar, DEU, 101–108.

[30] James T. Kajiya. 1986. The rendering equation. *SIGGRAPH Comput. Graph.* 20, 4 (Aug. 1986), 143–150. https://doi.org/10.1145/15886.15902

[31] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[32] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384. https://doi.org/10.1111/j.1467-8659.2009.01377.x _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01377.x.

[33] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (June 2013), 487–498. https://doi.org/10.1145/2508148.2485964

[34] Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M. Aamodt. 2021. Intersection Prediction for Accelerated GPU Ray Tracing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 709–723. https://doi.org/10.1145/3466752.3480097

[35] Lufei Liu, Mohammadreza Saed, Yuan Hsi Chou, Davit Grigoryan, Tyler Nowicki, and Tor M. Aamodt. 2023. LumiBench: A Benchmark Suite for Hardware Ray Tracing. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 1–14. https://doi.org/10.1109/IISWC59245.2023.00011 ISSN: 2835-2238.

[36] J. David MacDonald and Kellogg S. Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 3 (May 1990), 153–166. https://doi.org/10.1007/BF01911006

[37] Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M. Aamodt. 2022. Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 263–281. https://doi.org/10.1109/MICRO56248.2022.00027

[38] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 173–174. https://doi.org/10.1109/MSE.2007.44

[39] Jundong Tan, Zhuo Su, and Yunliang Long. 2015. A Full 3-D GPU-based Beam-Tracing Method for Complex Indoor Environments Propagation Modeling. *IEEE Transactions on Antennas and Propagation* 63, 6 (June 2015), 2705–2718. https://doi.org/10.1109/TAP.2015.2415036 Conference Name: IEEE Transactions on Antennas and Propagation.

[40] Aditya Ukarande, Suryakant Patidar, and Ram Rangan. 2021. Locality-Aware CTA Scheduling for Gaming Applications. *ACM Transactions on Architecture and Code Optimization* 19, 1 (Dec. 2021), 1:1–1:26. https://doi.org/10.1145/3477497

[41] Vulkan. 2024. Home | Vulkan | Cross platform 3D Graphics. https://vulkan.org/

[42] Ingo Wald. 2011. Active thread compaction for GPU path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG '11)*. Association for Computing Machinery, New York, NY, USA, 51–58. https://doi.org/10.1145/2018323.2018331

[43] Kathryn Williams, Luis Tirado, Zhongliang Chen, Borja Gonzalez-Valdes, José Ángel Martínez, and Carey M. Rappaport. 2015. Ray Tracing for Simulation of Millimeter-Wave Whole Body Imaging Systems. *IEEE Transactions on Antennas and Propagation* 63, 12 (Dec. 2015), 5913–5918. https://doi.org/10.1109/TAP.2015.2486801 Conference Name: IEEE Transactions on Antennas and Propagation.

[44] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 76–89. https://doi.org/10.1145/3503221.3508409